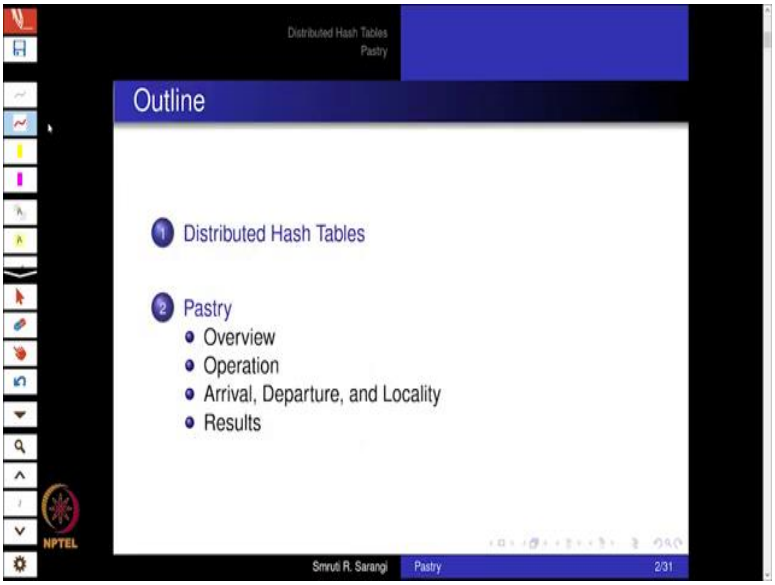
Advanced Distributed Systems
Professor Smruti R Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
Lecture 03
Pastry: A distributed hash table

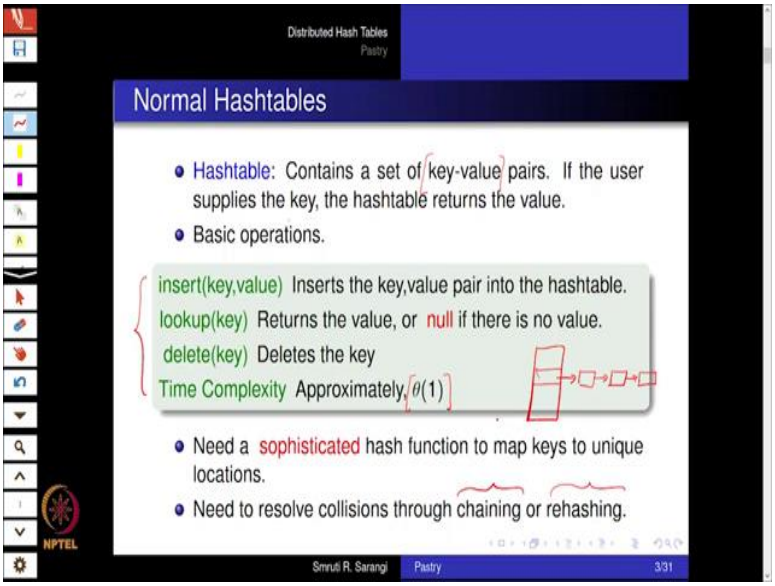
Welcome to the study of distributed hash tables. Distributed hash tables are by far the most commonly used data structures in distributed systems. Many commercial systems are built on top of distributed hash tables. Pastry is one of the most common designs in this space. And as we shall see later in the course, many, many designs are built on top of Pastry or a Pastry like system.

(Refer Slide Time: 0:47)



So a brief overview of this slide set, we will start with describing what distributed hash tables are? and then we will move to describing the specific operations of Pastry. So we will discuss in a broad overview, the operation of Pastry, arrival departure and locality of nodes, and finally the results.

(Refer Slide Time: 1:11)



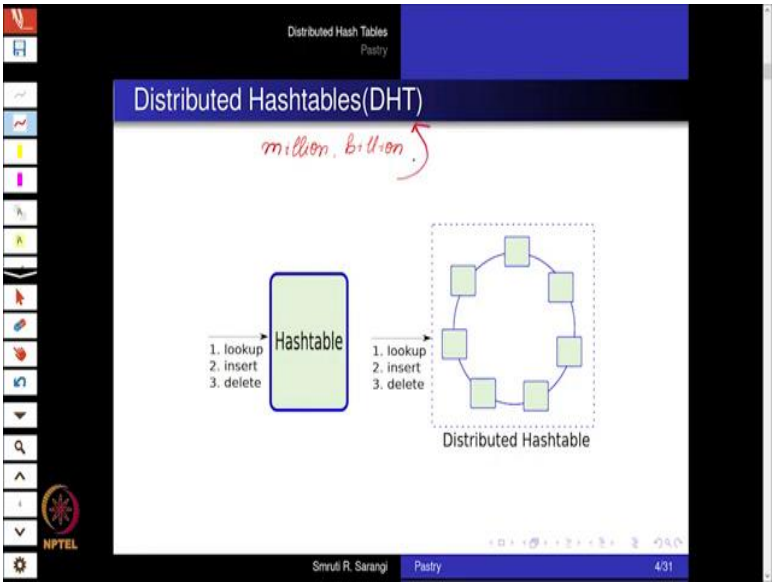
So before describing what is a distributed hash table, it is important to describe what is a normal hash table? So any student of data structures would have encountered this term before, so there is nothing new over here, a hash table in this context is also the same as what we have over there. It is essentially a dictionary that contains a set of key value pairs, that is the important operative term over here.

It is a set of key value pairs where the user simply supplies the key and the hash table returns the value, as opposed to an array where we index based on an integer, here we send the key and then we read the value. So there are several functions that a hash table supports. It supports the insert function, insert key value, inserts a key value pair into the hash table. There is a lookup operation. So for a given key it either returns the value or returns null if there is no value.

A delete key operation where we delete a key and approximately all of these operations take a constant amount of time, approximately constant, theta 1 amount of time. So the design of most hash tables requires a sophisticated hash function to map keys to unique locations and essentially reduce the chances of collisions. If there are collisions there are two standard ways of dealing with them, they are known as chaining and rehashing.

So chaining basically means that we, essentially every entry of the hash table is a linked list and we traverse the linked list. Rehashing means that if one entry is full, then we search for an, in an alternative location, that is rehashing. So a distributed hash table is the same.

(Refer Slide Time: 3:24)



It is of course, much bigger. Well, how big? So consider all the movies the Netflix stores, it might be millions of movies. So if millions of movies are to be stored they cannot be stored on one machine. They have to be stored on thousands of machines and these thousands of machines also have to be geographically distributed.

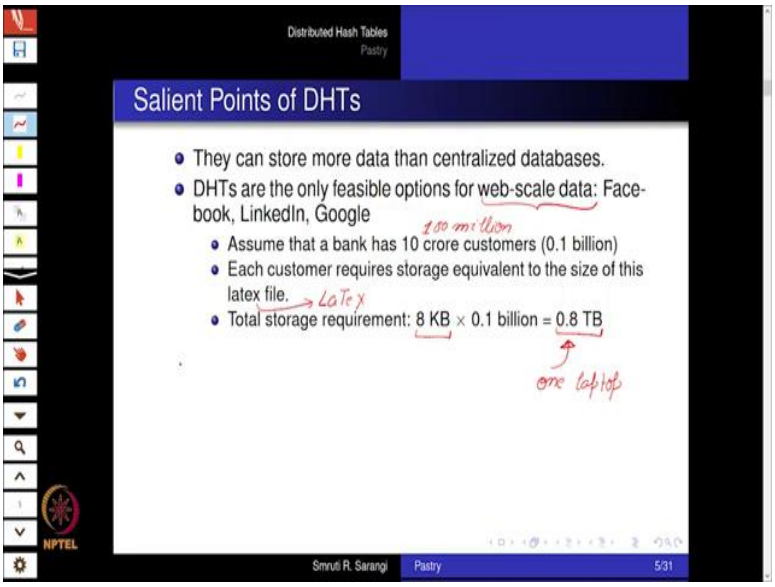
Because if there are not, if they are not distributed and there is a power failure in one geographic location, then what can happen is that the entire set of movies will go and in this case your favorite provider Netflix will not be able to operate. Hence, a distributed hash table is not meant for few entries like ten hundred and thousand it is meant for a million or billion entries.

So consider Amazon, just look at the share number of items that Amazon actually lists on its store. So if I were to have a quick search operation, in the sense let us say I am interested in the coffee mug and if I just want to, if I type coffee mug and I would like to see all the matching entries for coffee mugs, then I would need a hash table of sorts and that is why this forms the key search structure in almost all major cloud infrastructure, major cloud-based systems.

That includes Amazon, Netflix a large part of Google, LinkedIn, Facebook as we shall see in the third part of the course. So in a hash table we have three operations, lookup, insert and delete. So what we do is we will see this as a recurring theme that we typically organize the nodes that are storing the key value pairs in a circular format.

So this is essentially a virtual network, a virtual network is also called an overlay, so regardless of how the real network is we organize these nodes in a virtual circular form known as an overlay. And we will see this makes our job of locating where a key is substantially easier.

(Refer Slide Time: 5:49)



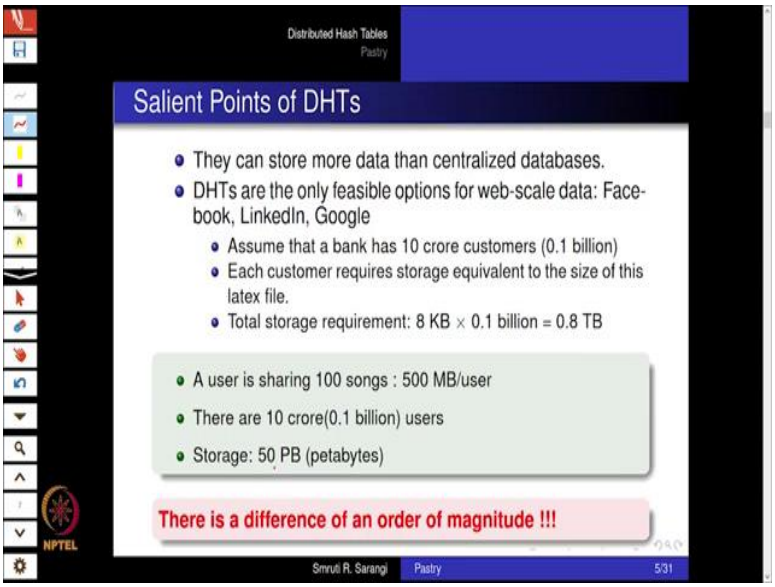
Why do we say that we need DHTs? Well, we need DHTs as we have discussed to store web scale data, which means a lot of data. So let us consider an example. So assume a bank has 10 Crore customers, the Crore is an Indian term. So roughly a Crore is 10 million, so it is 100 million customers. let us assume that the storage space that is required for each customer to store his bank account is the same as the size of this latex file.

So incidentally this entire presentation was made using latex, so the latex should rather be represented like this, it is a more correct form of doing it. So what is latex? It is a wonderful type setting system. I have used the beamer package, so would encourage all of you to take a look at that. So I also took a look at my bank accounts and I took a look at all the transactions, I put them in a text file.

So the size of all of that was roughly the size of this file. So this file is around 8 kilobytes and 8 kilobytes times 0.1 billion entries is 0.8 terabytes. If you would see this is not much, so a regular desktop processor or even a combination of pen drives can provide you this much of space. So definitely one desktop processor or even one laptop, a modern laptop these days can provide you 0.8 terabytes of space.

Incidentally, the laptop that I am using to record this video has half terabyte of space, so this basically means that if I have 100 million customers in a bank, which is not a small number by any means, the total data requirement is not much as opposed to web scale data where I would actually need much, much more. So that is the reason for a traditional banking application, I can use a traditional database but for a much larger source of data I would need a DHT.

(Refer Slide Time: 8:16)

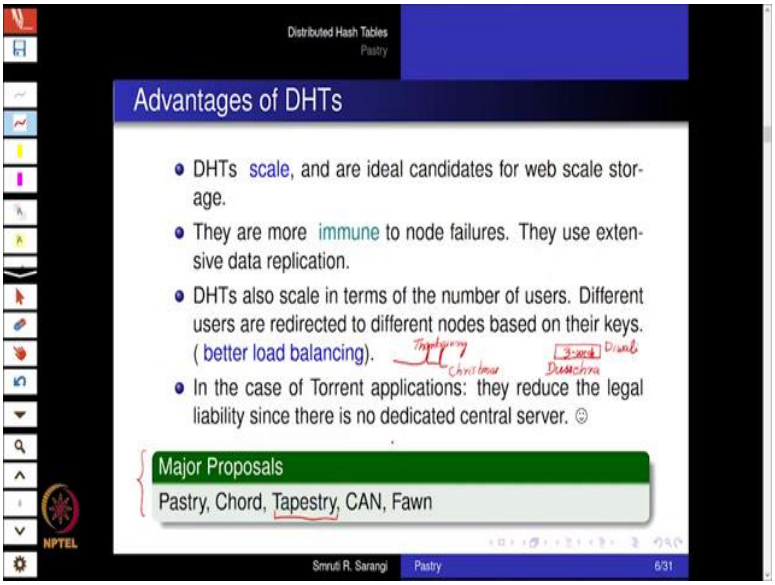


So how large are web scale data? How large is web scale data? So let us say user on an average shares 100 songs, assuming an MP3 format, it will be 500 megabytes of space per user, if there are again 0.1 billion users so it is easy to do the math. We will have 50 petabytes of storage, which is way more than this number. And it is also not possible to have that much of storage at a single location.

Number one, that well it is a huge amount of storage so the operating costs, the cooling costs will be very high, and second, if let us say there is a power outage the entire system is gone, see, we would like to have a large system consisting of a large number of machines geographically distributed that can provide this much of storage and this is a significant, even in today's technology this is a lot of storage.

So the main aim here should be that for web scale data we provide tons of storage and this storage in a certain sense is used to provide web scale services. So a difference is clearly an order of magnitude and this is something we need to bear in mind.

(Refer Slide Time: 9:45)



So DHTs scale, they are ideal candidates for web scale storage. Furthermore, they are immune to node failures in the sense if there is an outage in one region, it is okay. So this happens all the time, servers have an outage all the time, but we never really see Google, Amazon, Netflix, etc. actually go down, primarily, because they rely on technology that is immune to at least localized node failures.

Furthermore, the scale in terms of number of users, so here is one fun fact, so particularly in the U.S. most of the shopping happens, if I were to draw a graph of shopping most of this shopping would happen, there are two peaks, around two times, so one is Thanksgiving and the other is Christmas, so that is when most of the shopping actually happens and the rest of the time the amount of shopping is not much.

See, if I were to see in India, most of the shopping would actually happen now when I am recording the video, which is roughly a 3 to 4 week time, so it starts with the festival Dussehra and then it pretty much ends with Diwali. So this is a three week window, so this is where most of the shopping happens in most of India.

And so, in this time frame if let us say I am an ecommerce provider, I would need a massive amount of compute and storage, not for the rest of the times. So what we would want is we would want a system that can kind of that, is sense fluid, whenever I want to add new servers it will be very easy to add new servers and once the rush hour is gone, then I can remove the servers.

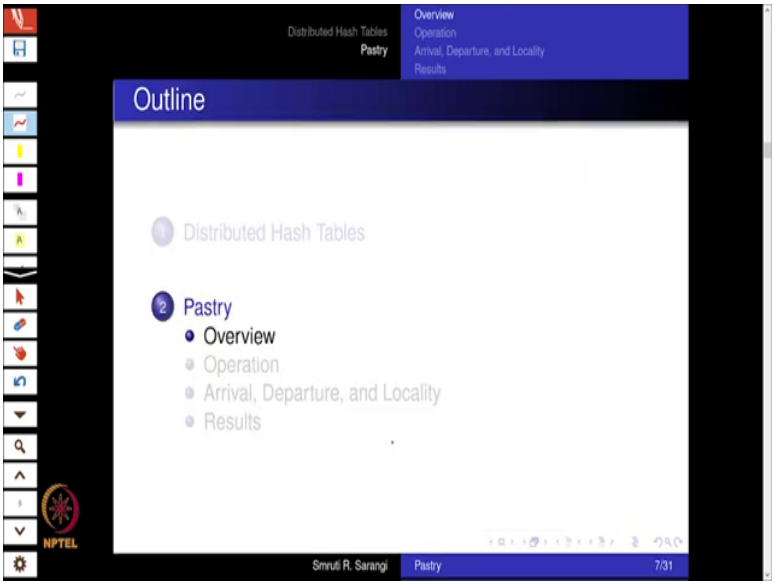
For example, Thanksgiving is roughly in October, November timeframe festival, and Christmas is December 25th. And whereas, Diwali and Dussehra are September, October time festival. See, if I am running a global ecommerce system then it is very easy, I can add some servers during Diwali, Dussehra and again remove a few of them, again add them during Thanksgiving, then remove a few of them, again add them during Christmas.

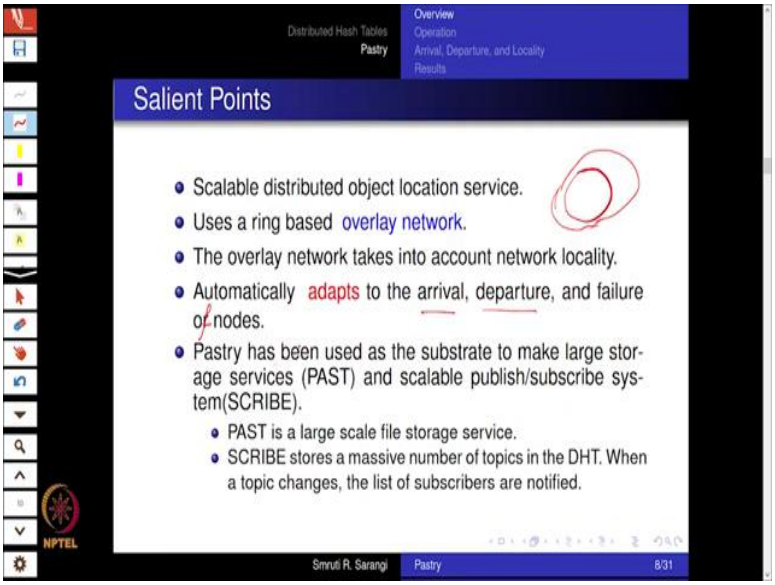
So what we need is we need some amount of flexibility with adding such servers. Furthermore, many of you would have used a BitTorrent like applications. So BitTorrent unfortunately even though it is great technology and that led to the DHT technology mature in, sadly many users do put illegal videos on the Torrent servers.

So, having a distributed DHT does reduce the legal liabilities, but again that was never the intended use. And just a word of caution, please, please, do not access, download, post illegal videos, videos for which you do not have a license on any kind of a BitTorrent system. The torrent does use a DHT that is how it works, that is another great example of DHT but that should not be used to encourage piracy because piracy kills the movie industry.

It kills the music industry and since all of us like music we would like musicians and filmmakers to be paid as well. So what are some of the major proposals in the world of DHTs, well, we have Pastry, Chord, Tapestry is something that BitTorrent uses not exactly, but systems are similar; CAN and Fawn, so these are some of the major DHT related proposals that are there.

(Refer Slide Time: 14:01)



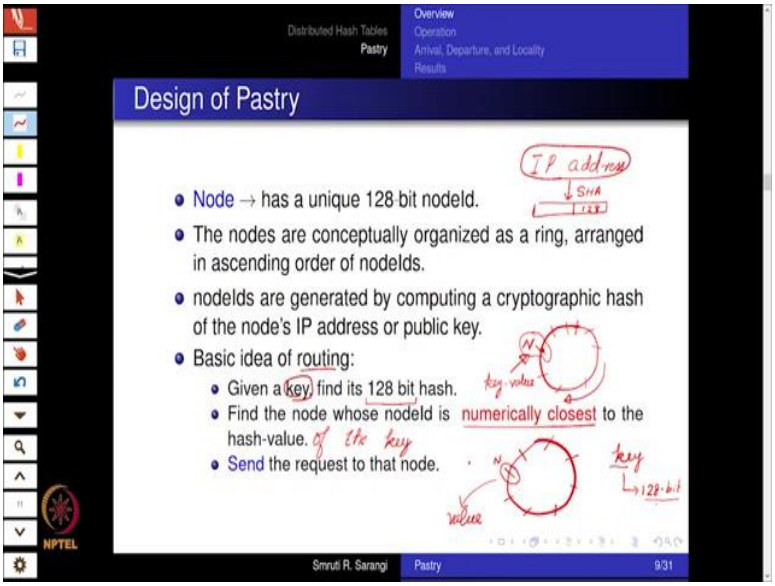


A brief overview of Pastry. So pastry is a scalable distributed object location service where pretty much we store objects in a key value format in a DHT. We use a ring based overlay network, which means that it is an, overlay is a network over a network, it is a virtual Network. And regardless of where the servers are physically there we just assume they are logically arranged as a ring.

This network does take into account locality in the sense that some amount of locality between the nodes is taken care of, we will see how. So the locality means a physical proximity, some of it is taken care of, but it is important to understand how. Furthermore, this is a very flexible ring in the sense it can shrink, it can grow, if we want to add new servers the size of the ring will grow, it can become much bigger.

And so basically it is rather flexible in that sense. It adapts to the arrival departure and failure of nodes. So this should be off over here. Pastry has been used by some popular storage services like PAST and SCRIBE. So PAST is a large scale file storage device and SCRIBE is where we publish and subscribe to topics. But as I said it was one of the earliest DHTs and it has in a very positive way influence the development of DHTs quite a bit.

(Refer Slide Time: 15:54)



So let us now without further ado discuss the design of Pastry. So we assume that every node is associated with the 128 bit node ID. So the node ID can be generated as follows, we can for example, take the IP address of the node and we can apply a certain transformation to the IP address. Like for example, we can encrypt the IP address using the SHA algorithm, so then we will get a piece of encrypted text.

And in the encrypted text also known as a cipher text we can extract the lower 16 bytes. The lower 16 bytes is essentially 128 bits. So these 128 bits can be extracted and so this is how we can give every node a unique 128-bit node ID? So now the aim is to conceptually organize these nodes as a, ring where we assume that they are arranged in ascending order of node IDs.

So, let us see if we were to walk clockwise. So we have a node here in ascending order, we will have nodes, so it is kind of a conceptual ring where they are done but of course, ascending order will happen and there will be one point of discontinuity which is the beginning, we are okay with that much, but otherwise they will be arranged in a, they will be arranged in ascending order.

So node IDs are generated by computing a cryptographic hash of the nodes IP address or a node can have a public key that could be the 128-bit node ID. So what is the basic idea of routing here? Well, given a key, the basic idea, so we are not coming to routing, we are not looking at the design, the design has already been mentioned that we have a set of nodes, each node has a 128 bit node ID and it is organized conceptually as a ring.

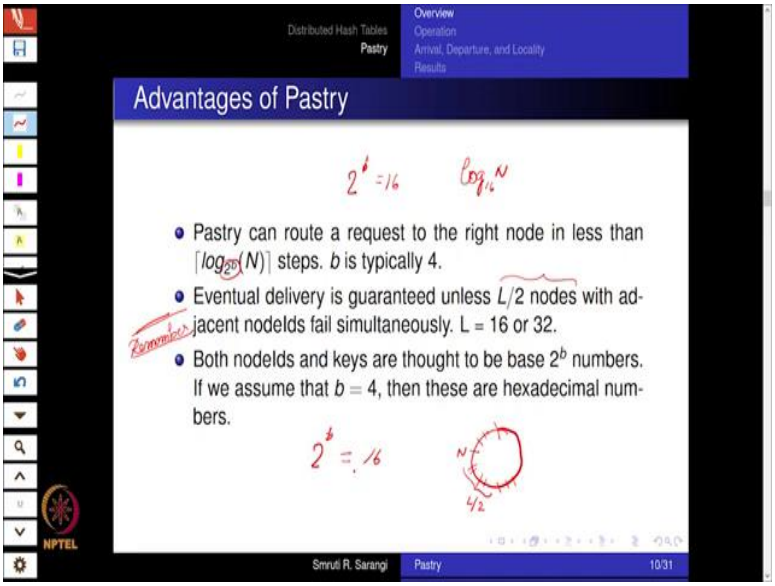
And we can assume without loss of generality that as we walk clockwise the node ID is increase. So how do we locate a key over here? Well, for the key we do the same. We compute a 128 bit hash, so let us see that this is the, these are the nodes, the cuts are the nodes. So we need to find that node ID, which is numerically closest to the hash-value, hash-value of what? hash-value of the key.

So what we do is that we need to walk this ring and find that node ID, which is the numerically closest to the hash-value of the key. So let us see it is this node, so then what we do is that this node ID, this node with this ID is automatically responsible for storing the key value pair. Which key value pair? Well, the key value pair off this key.

So what is the broad idea? So just think about it. This is the core idea of DHT. So I will go over this several times, it is important for the viewers of this video to get the idea. So the idea is that we have a circular ring of nodes. How is it done? Well, it is a conceptual ring, it is not a physical ring, where we assume that nodes are organized in increasing order of their node IDs. Now given a key we need to find the node that contains its value.

So what we do is that from the key we create a 128 bit node ID and this number is used to find the node ID within this ring that is the numerically closest to it, minimizes the absolute value of the distance and let it be this. If it is this we send a request to this node, provide the key it will provide us the value. As simple as that nothing more nothing less. This is the straight cut simple idea of a DHT and almost all major commercial systems use some variation or modification of this.

(Refer Slide Time: 20:26)



So what is the advantage of Pastry? So why is Pastry great? So let us say we have n nodes, so what Pastry claims is that in less than log n to the base 2^b L, come to what that is, but let us assume that $2^b = 16$. So, I will discuss this later in great detail, but less than $log_{16}N$ steps Pastry can route the message, the message meaning the key can be sent to the node ID.

So we will discuss in detail what is a step? But let us say step is sending one message, so with less than login to the base 16 messages, we can find the correct node and we can send a message to it and what will that message be, that look this is the key what is the value. And so this, instead of 16 it can be 8, it can be 32, but 16 is most commonly used. And so, but because it is a power of 2 that is the reason I have written 2^b.

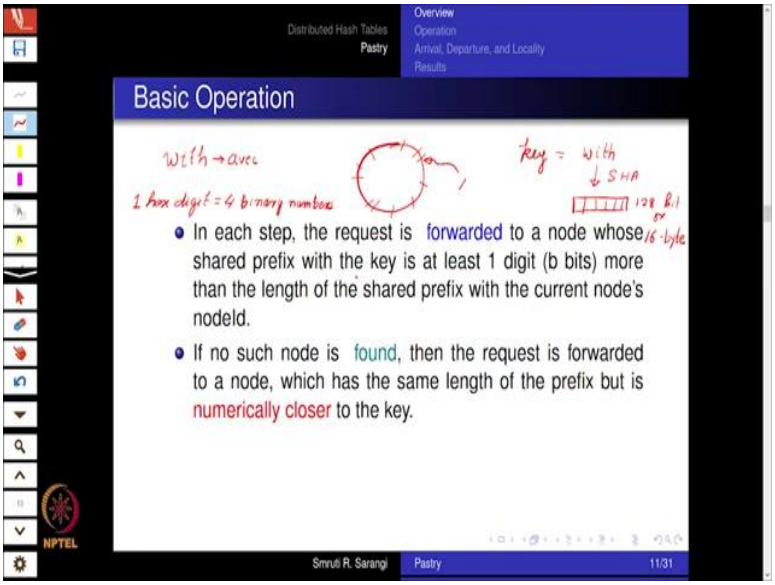
If B is typically 4, 2^b= 16. Pastry does a wonderful job in tolerating node failures. Nevertheless, eventual delivery is guaranteed at the destination, unless L/2 nodes where L is a user defined constant with adjacent node IDs failed simultaneously L can be 16 or 32. So what this is basically saying is that look if this is the final node and let us say there are many, many node IDs that are different points of the ring.

And similarly it is over here, so if L/2 nodes with adjacent node IDs, if they fail, then only we can say that we might not reach the destination. But if it is never the case that L/2 adjacent nodes fail, then we will reach the destination all the time. So well, why is this the case? Not now, I will tell you later, but this can be remembered.

This can be a point worth remembering that there is a certain amount of immunity to node failure that even if nodes fail we can still guarantee delivery at the destination, of course, assuming the destination has not failed subject to the fact that L/2 nodes with adjacent node IDs have not failed. One important disclaimer, both node IDs as well as keys have the same base. Well, again they need not have, but there are many advantages if they do.

So both are assumed to be 2^b numbers. Well, B is typically 4, so we will assume that 16, 2 raised to the power B is equal to 16.

(Refer Slide Time: 23:41)



So what is the basic operation? Well, the basic operation is like this, that in each step the request is forwarded to a node whose shared prefix with the key is at least one digit mode than the length of the shared prefix, so the current node's node ID. This is a very loaded statement. I would suggest that we go slow. And so you remember this, but I will keep visiting, revisiting this quite a bit till the idea is kind of clear.

So, let me show this, so this is the key idea and I would suggest that this slide which is slide number 11 is seen several times till the idea is completely understood. So the idea is that we have nodes in the node ID space and the point where I am drawing, so they need not be in a uniformly placed, they can be non-uniformly placed as well; given that the node ID is coming from a random process.

So what we do is that let us say I have a key and I want to figure out the IP address of the node within this ring that contains the key and consequently its value. So let us say that this is a

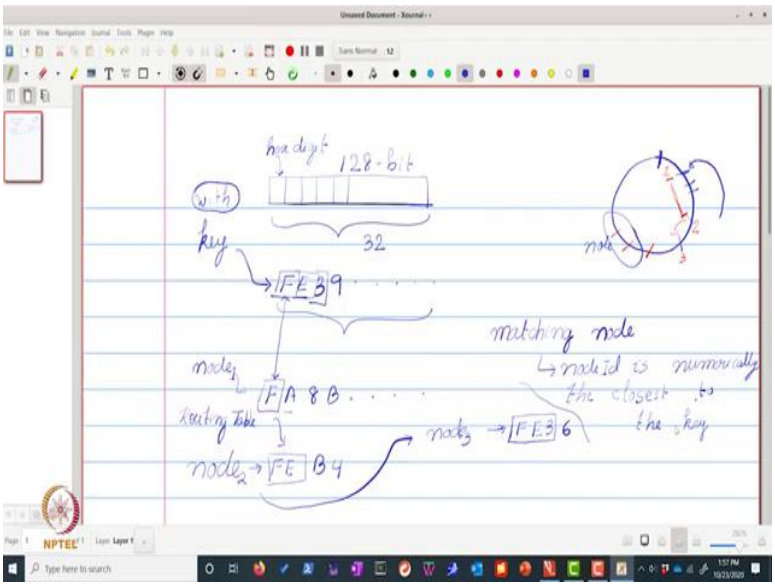
regular, okay let us say that it is an English to French dictionary. So if it is an English to French dictionary, so let us say the word that I want to search for is ‘with’ and the French of ‘with’ is avec.

So I want to find out where ‘with’ is in this entire DHT. So what is my key here? It is ‘with’. What now I do is I run it through an ID generator which as I said is typically the SHA algorithm which is used. It will give me a 128 bit or 16 byte. So it will provide a 128 bit or 16 byte number and then what I will do is that I will go to any node which I know to be a part of the ring and I will give it this key.

So let us say the first four, so let me now divide this into hex digits. So what are hex digits? Well, hex digits are hexadecimal digits and each hex digit is 4 binary numbers. So this is important. 1 hex digit is 4 binary numbers, if you are not able to understand why this is the case, then pick up a book on Boolean Logic and first figure out why this is the case and then you proceed.

If you do not understand this you will not be able to understand what happens next. So 1 hex digit is 4 binary numbers, so I need to break it down to a sequence of hex digits and search for it. So let me now shift to another software, where I can show this process particularly in great detail.

(Refer Slide Time: 27:28)



So what did I say? I said 1 hex digit is 4 binary numbers and that needs to be understood. So given the fact that I have a 128 bit hash, I can divide it into 4 bit numbers. How many such 4

bit numbers will I have? I will have 32 such 4-bit numbers. Each one of them is a hex digit. Each one of them is a hex digit.

So let us assume that for the key, what was my key? well, it was the English word ‘with’ and this key; the first few digits are like this, so these are all hex digits by the way. And then there are a few more digits will that I initially do not care. So what I would do is that given that this is the ring, I would know some node within the ring. So let us say this is a starting node.

I would know some node within the ring and I will send a message to that node and I will tell the node, Hey look, this is my key, of course, I will send a sequence of 32 hex digits, look this is my key, kindly find the node that contains the key. So what that node will do is like this? That node will check its own node ID, its own node ID might be something of the form, it does not matter.

So it will match hex digit by hex digit between the key and itself. So in this case, it will find that the first digit matches, which is a good thing. Given the fact that the first digit matches it is actually great. So then what it will do is that it will take a look at the next digit. So what is the next digit in the key? Well, what is the next digit in the key is ‘E’ and in this case it is ‘A’.

So this is technically speaking, are not desirable, because we would like to have a match. So given the fact that these two digits do not match, what the node will then try to do is, it will try to find another node where the first two digits match with the key. In the sense, so let this be node 1, it will search its own data structures. So the data structure that it will search is called a routing table, that is the data structure it will search. In that it will try to find.

If it knows any other node for which the first two digits match, in the sense the first two digits are F and E, so let us assume it is able to find one such node, so we had initially approached this node. So let us say after, so let me change the color, we are initially approached this node and let us see now it forwards the message to this node, which is one message forwarding.

And here let us assume that this is node 2. With node 2 assume that the first two digits match ‘F’ and ‘E’, but the remaining two digits of the node ID are, well, the same thing, given that the first two digits are matched. We then search for the next match, so the node 2 will be asked at look the first two digits match with the key, it is ‘F’ and ‘E’. Do you know of another node where the first three digits will match?

So let us assume it knows. So in this case node 2 will further forward the message to node 3. That can have a node ID of the form ‘F’, ‘E’ then ‘3’ and then maybe this is ‘6’. Well, in this case we have increased the match, so now the first three hex digits match and the fourth one, of course, does not match, so then we will proceed in a similar manner.

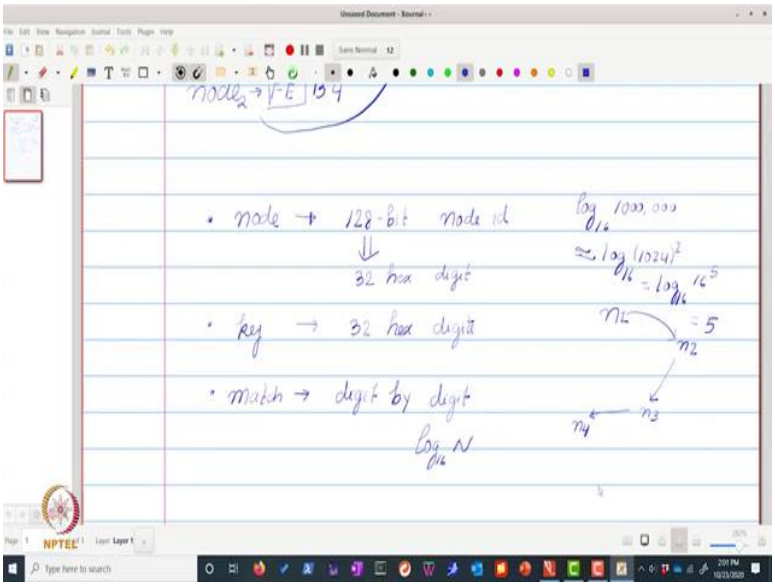
So what will happen is that maybe this was node 1, it forwarded it to node 2, now more digits match, then again it forwarded it to node 3, now again more digits match and then node 3 will forward it to another node and another node and another node and ultimately what will happen is that we will have a fair number of digit matches.

So we will, given the fact that it is 128 bits and that is a very large number, this process will terminate at some point. So we will mathematically see when and where? but let us assume that a certain number of digits match and after that we are not able to find a match. Then we will discuss one more algorithm of how to search the neighborhood to find the node that contains the key. But just look at the way that we are constructing these matches.

So we have started from left to right, so at least most of the MSB bits, the Most Significant Bits, they are matching, so which is kind of bringing us closer and which is bringing the node ID and the key in a certain sense closer and closer and closer and ultimately the target node and the key they will sort of start falling in the same neighborhood and then we will apply a different kind of search and ultimately find the matching node.

So recall what is the matching node? Well, the matching node is essentially that node whose node ID is numerically the closest the closest, so it is numerically the closest to the key, so as we are matching these MSB hex digits, we are getting closer and closer and closer, till they start lying in the same neighborhood and then we can apply a different algorithm and then perform the match. So what is the key important broad idea?

(Refer Slide Time: 35:09)



Well, the broad idea, let me summarize this. This is something that should be summarized. So, if I were to write the main points we take a note, we convert the nodes, identifier, whatever it is, like the IP address, to a 128 bit node ID. So that is not that important in this context in the sense that here since we are looking at a base 16 representation we would like to look at this as 32 hex digits. And what is the relationship between a binary bit and a hex digit?

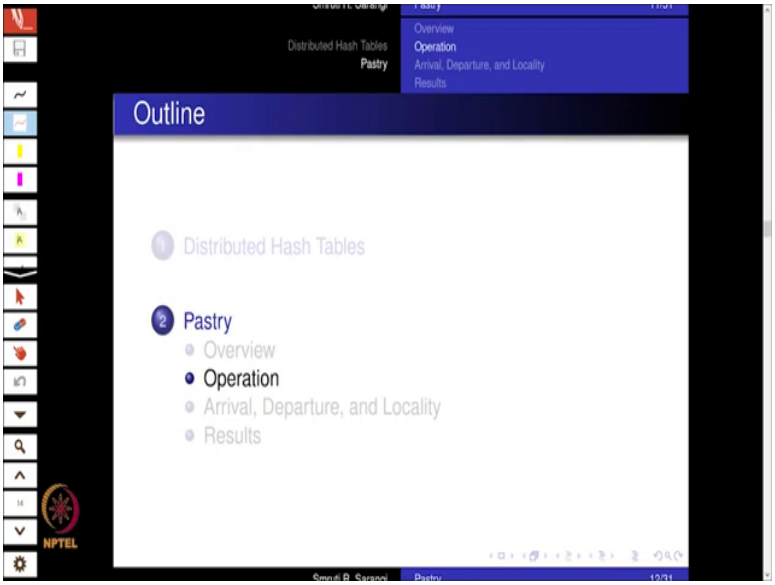
Well, if you do not know then you should not be taking this course, first recapitulate Boolean algebra and then come back. The second is the key. Well, the key also we do the same, we have 32 hex digits and then we start from the left as we did and then essentially we do a process of matching, where essentially we match digit by digit. Well, how do we match it?

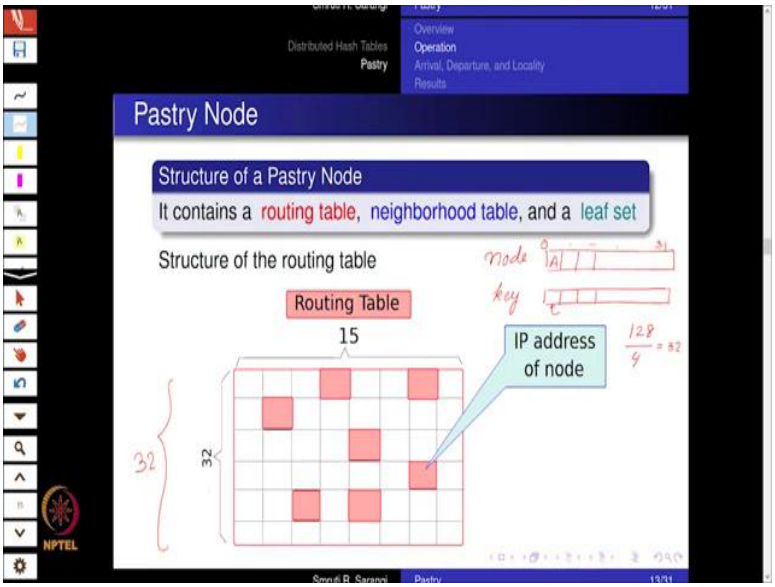
Well, the way that we actually match is that; well, we start from the left, we start matching the digits and if at a given node if only two digits match to make the third digit also match with the key, essentially one node forwards the message and node N1 forwards the message to node N2 to increase the amount of matching, N2 then forwards it to N3, so on and so forth till we reach a neighborhood and there as I said we apply a different kind of search.

And ultimately we find that node, which minimizes the distance with the key and furthermore, our claim is that this is being done in less than $Log_{16}N$ steps or messages. So if we look at this even if N were to be a very large number, let us say n is 1 million. So then let us compute log_{16} 1000,000. So this is nothing but, so let us approximate that as $log_{16}(1024)^2$, which is equal to log of

1024 is 2 raised to the power 10, so this is 2 raised to the power 20. So if it is $log_{16}\ 16^5 = 5$. So what we see is that even if we have a very, very large space off nodes and it does not matter, the routing can proceed quickly, rather very quickly. The reason for that is that we reduce the search space exponentially in each step.

(Refer Slide Time: 38:31)



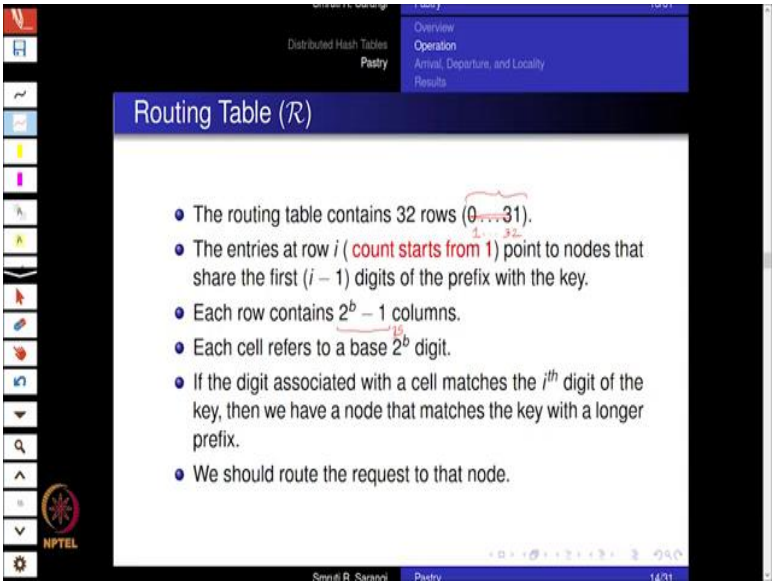


Let us now discuss the operation of the key data structures in the Pastry protocol. So the most important data structure is a routing table. So a routing table in this case where we are considering a base 16 system, it will have 32 rows as we see over here and it will have 15 columns. So why 15 columns and 32 rows if it is not self-evident by now? Well, if we consider two digits, so the two hex digits, then we actually have 15 possible choices.

So let us consider the first digit, say the first digit is let us say A, then well, if the first digit matches that is good, otherwise if it does not match then we have 15 possible choices over here and we have one column for each choice. So this is why we have 15 columns in each row. Second, given the fact that one hex digit is 4 binary bits.

Again I said that the viewer should be in a position to understand why this is the case? if not go back study this and come back. So since there are a total of 128 bits and one digit is 4 binary bits, so 128/4 = 32. So we will have 32 hex digits in each ID, either the node ID or the key ID. So at the most we can have 0 bits matching till 31 bits matching, so hence, we will have 32 rows.

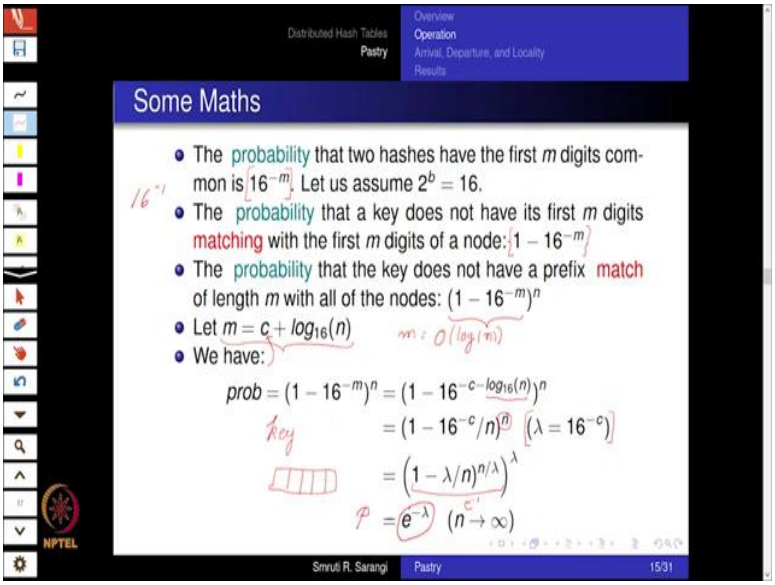
(Refer Slide Time: 41:02)



So the routing table thus contains 32 rows, let us number them from 1 to 32. So the entries at row i where the count is starting from 1, essentially point to all the nodes that share the first (i – 1) digits of the prefix, (i – 1) digits of the node ID with the key.

For example, if it is the first row, then what it means is that the first digit itself does not match, so the number of digits that are matching are actually 0. And so then we try to match the first digit and let us say the first digit is over here, then we go to IP address that points to the first digit, so on and so forth. So this has been explained that each row contains $(2^b - 1)$ columns, if say B = 4, this is 15 and the matching process has also been explained. So the only correction in this slide is instead of 0 to 31 it will be 1 to 32.

(Refer Slide Time: 42:26)



So let us do a little bit of math. It is important to do this math to understand how quickly the Pastry protocol will actually converge? So let us find the probability that the two hashes have the first m digits in common, so what is the probability? Well, the probability that two digits are common is 1/16 or 16^{-1}. So this follows from the fact that we are using a base 16 system.

So the probability that the first m digits are common is 16^{-m}. The probability that a key does not have its first m digits matching with the first m digits of a node is essentially (1 - 16^{-m}). The probability that the key does not have a prefix match of length m with all of the nodes in the system and assume there are n nodes in the system is essentially (1 - 16^{-m})^n.

Given that these are independent events, they will get multiplied, so it is this number to the power n. So what should m be? Well, let us set m = c + $log_{16}(n)$, so the logic of doing this will be clear very soon when we describe the rest of the mathematical expression, but let us assume that this is how we set it. So essentially m is, if I were to explain it in order semantics, m = O(log(n)).

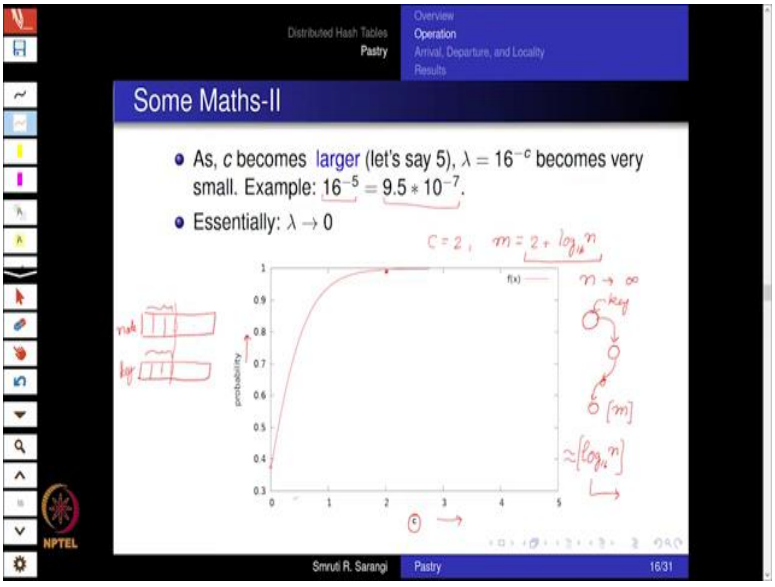
But let this be the exact expression for n, where we will discuss the importance of the constant c later. So the probability = (1 - 16^{-m})^n = (1-$16^{-c-log_{16}(n)}$)^n. So this is (1-$16^{-c}/n$)^n. So let us say, let the constant ($\lambda = 16^{-c}$).

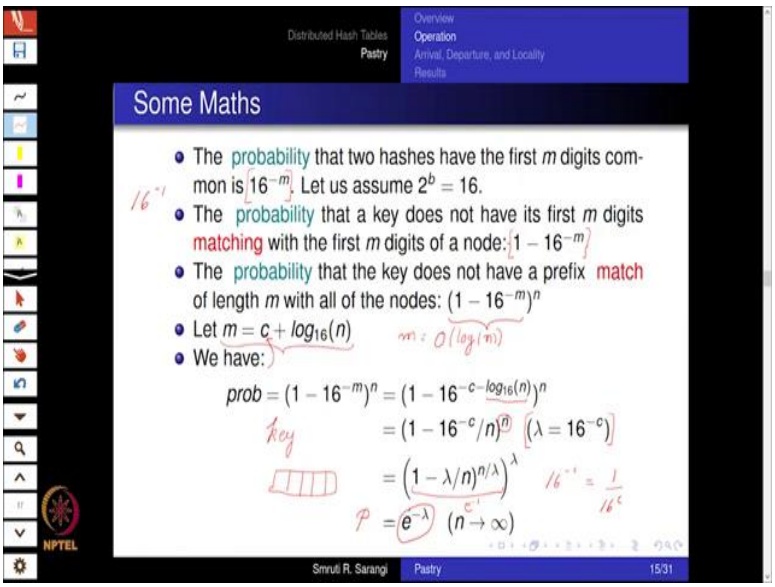
So given that this represents 16^{-c}, we can simplify this expression somewhat, I mean, quite a bit, so 16^{-c} = λ / n. the exponent can be converted to (1 – λ/n)$^{n/\lambda}$)$^{\lambda}$. So any multiplication and exponent will become a power. This is an expression that we have seen as n tends to infinity or as n becomes a very large number.

This is a very standard reduction; this number is equal to e. so this number is equal to e^{-1} or $\frac{1}{e}$. So this is an important point to keep in mind that this number over here is a, this is a known identity that as n tends to be a very large number, this expression can be approximated as e^{-1} and e^{-1} to the power lambda is $e^{-\lambda}$. So what is this probability again?

Well, this is a probability that look I have a key, I am interested in the first m digits of the key and the first m digits of the key do not match with any node of the system and this P = $e^{-\lambda}$. So let us graph this.

(Refer Slide Time: 46:41)





So this is where, so if you look at lambda, well, what is lambda? It is $16^{-c} = \frac{1}{16^c}$. So as C becomes larger, so let us say c becomes 5, so ($\lambda = 16^{-c}$), which means lambda itself will become very small, so 16^{-5} = 9.5 *10^{-7}.

So what I can do is that this probability that has been computed, this can be computed as a function of c. Essentially this probability is a function of c because lambda is a function of c. So if this is a function of c what I can do is I can have c on the x axis increasing, of course, and I can have probability on the y axis. Again you will find it increasing.

So what you basically get to see is that as we increase the value of c the probabilities, well, starts at around 0.35 and gradually saturates to 1 by the time that c = 2 and by 3 definitely c = 2 is good enough. So what this essentially tells us is that look if C = 2 then what will m be equal to? m = 2 + $log_{16}N$, so of course, the reason we have not considered n over here is because we are considering large n.

So if you are considering large n, this expression essentially says that, look for any the moment that m crosses, the moment that m = 2 + $log_{16}N$, after that the first m digits will not match with any other node and the probability of that becomes pretty much equal to 1. So the probability that you will not have a match with any node and the fact that your routing tables will be of no use that becomes almost certain.

So what is the key implication? Well, the key implication is that the number of routing tables that we actually need to jump, so basically what happens? So we have one node and then a message is sent to another node to increase the match, then again to another node, again to another node, and in every stage we increase the matches by one.

But what this result says is that the number of messages we need to send between the nodes, which is essentially the fact that a key arrives in one node, let us say it arrives in this node and then it is iteratively sent across the nodes with the match increasing by one digit across the nodes, so at the most we will send it m times and by the time we send it m times, what we will find is that after ≈ 2 + $log_{16}N$ steps our number of matches is not increasing.

What I can say in other words is that if I were to consider this number 2 + $log_{16}N$, approximately I will need these many steps in my protocol to arrive at a node beyond which I cannot route to other nodes because the match in terms of digits will not increase. I might have

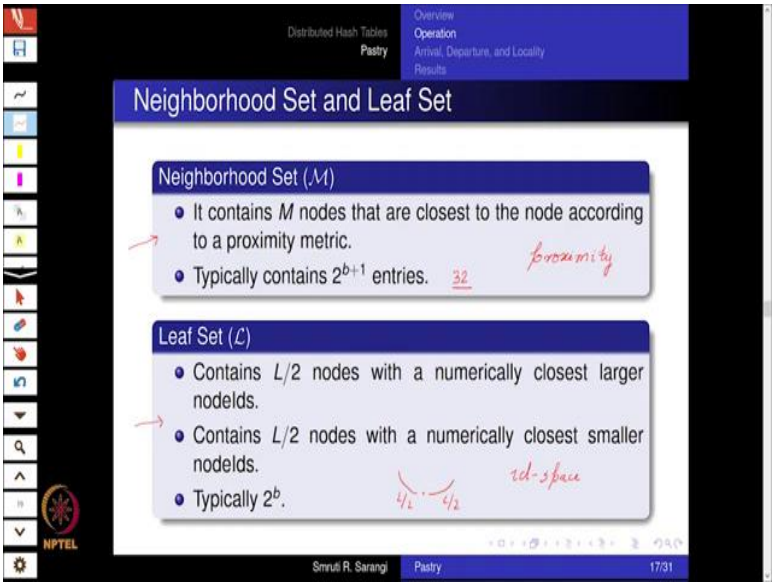
to do something else to find the numerically closest node, but at least the matching in terms of digits will cease after log n steps.

So this is kind of hinting or providing us some direction that most likely the message complexity of Pastry, the number of messages we need to send to reach the node that owns the key, the numerically closes ID is somewhat close to, very close to this number and Pastry is essentially a log n step protocol. So the login step protocol, most of it is clear from the analysis.

It basically says that look you send login messages, you will arrive at a node and which has the maximum amount of match with the key, from that note we need to find the node that is the numerically closest. So given the fact that consider both the node ID as well as the key ID. So given the fact that we are actually proceeding in the MSP direction and we are matching the top digits, we are already getting closer numerically.

But once we stop at one point, it is possible that we have a set of nodes that have the same amount of match in these digits, out of them we need to choose the node whose rest of the digits make it numerically the closest.

(Refer Slide Time: 52:13)

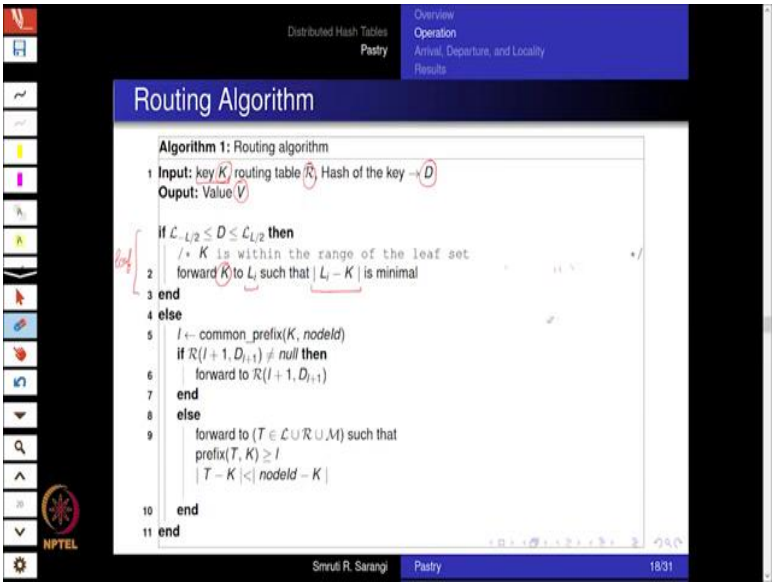


So this is where we will use two important data structures, two other data structures, the neighborhood set and the leaf set. The neighborhood set contains m nodes that are closest to the node according to let us say geographical proximity. So this will typically contain $2^b + 1$ entries, in this case 32 entries. So the neighborhood set can be thought of as a geographical neighborhood set.

So this can contain 32 entries. Furthermore, what we can do is we can have a leaf set where we typically have 16 leaves, so the leaf set for any node will be L/2 nodes on the ring with numerically closest larger node IDs and numerically closest smaller node IDs L/2 on this side, L/2 on the other side. So that is the leaf set. So the leaf set captures the proximity in terms of node IDs.

So this is a proximity in the ID space and the neighborhood set is a proximity in the geographical space. In the sense that if I have a node in my data center, other nodes in my data center are a part of my neighborhood, but they are not a part of my leaf set. So this kind of captures network proximity or geographical proximity.

(Refer Slide Time: 53:51)



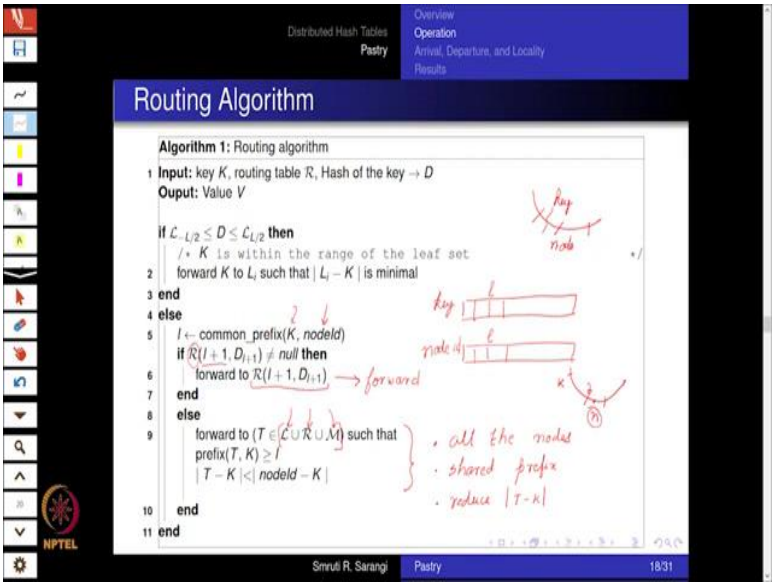
So keeping these in mind let us proceed, so this is the routing algorithm. So assume that we have a key K, a routing table R, let the hash of the key be equal to D. So the key important variables, I am just encircling, and let the value that I am after be V. So what I do is that first I check my leaf set. So this is the routing algorithm that every node will follow. So the first thing that you do is you check your leaf set, in the leaf set if it is between the first and last leaf.

So let this be equal to $L_{\frac{L}{2}}$. Let this be equal to $L_{-\frac{L}{2}}$, which essentially means that we have L leaves on this side L leaves on the other side, so for a given node, so I am at a given node, so if the given node's node ID is this, I will essentially check my leaf set. If the key falls anywhere in this leaf set.

So we always assume that the leaf set information is accurate, so the key falls anywhere in this leap set then we know for sure which node is a numerically closest to it, because our view of this region is absolutely correct. So what we can do is we can forward the key K to the leaf Li, such that the distance Li - K is minimal, so this itself is the closest node ID.

So if I were to explain this in a different way, what this essentially means is that given if I am close enough to the actual node I will come to another node such as the key lies within the bounds of its leaf set. So within that I will always find the node that is the closest to it. So this is an important concept, but I would suggest the readers to go through it several times.

(Refer Slide Time: 56:37)



So now, let us look at the more common case where it is not within the range of the leaf set. So in this case let me find the length of the common prefix between the key K and the node ID. So this is not hard to find at all, all that we do is we take a look at the key. And then what I do is I take a look at the length of the common prefix which is the number of matching digits. So then let it be equal to L, I would like to increase L to L + 1.

So what I do is I take it in the routing table, I look at the L + 1'th row and I look at the next digit of the key, so this is what I would like to match. So in L + 1'th row I take a look at the next digit and if this is not equal to null, which means that I know of some other node, where a matching digit exists. Then I forward it to R l + 1, I forward it to that node. So this is a forward, and I essentially forwarding and the number of matching digits is increasing by 1.

If it does not, if I do not find such an entry in the routing table, then what I do is that I look at all the nodes that I know, all the nodes for whom I have information within my leaf set, within my routing table, within my neighborhood set. Out of all the nodes I choose a node such that the length of the shared prefix is, of course, $\leq$ l and essentially D - K, which is the distance is less than node ID - K.

So I forward it to any node that minimizes the distance between the node ID and the key. I can always do better in the sense that I can forward it to that node that minimizes the distance instead of any node where the distance is just lower than my current distance, but it actually does not matter. As long as the distance is decreasing I am good.

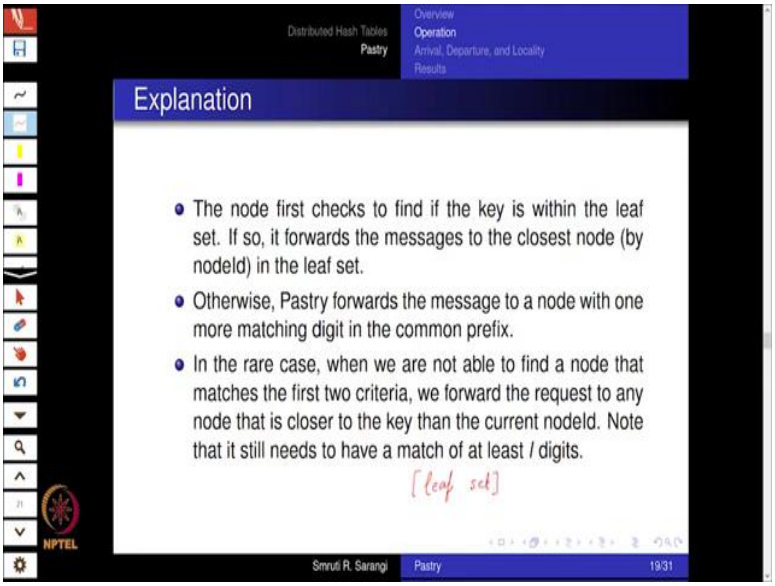
So what is the key idea here? Well, the key idea over here is that I take a look at all the nodes, for which I have information. So that means I will scan all my data structures and of course, the length of the shared prefix should be the same if not more. So I will not compromise on that, so the shared prefix, the length of that has to be maintained. So I will never compromise on that and then essentially I will reduce the distance between the node ID.

In the sense it is node T and the key K. So, this process is, of course, guaranteed to converge because what will happen is that if this is the current node ID, let us say on the ring, and this is the bounds of my leaf set, even if my routing table is not populated and the key is far away, at least what I will do is that I will find some node that is closer to the key.

If I am not able to find I can always choose the edge of my leaf set because that is guaranteed to be closer to the key than the current node n. So, the current node n, of course, has some distance with the key, but I can always go closer by essentially looking at the boundary of my leaf set, the node which is at the boundary of my leaf set and forwarding the message, passing on the message to that node. So that is guaranteed to be closer.

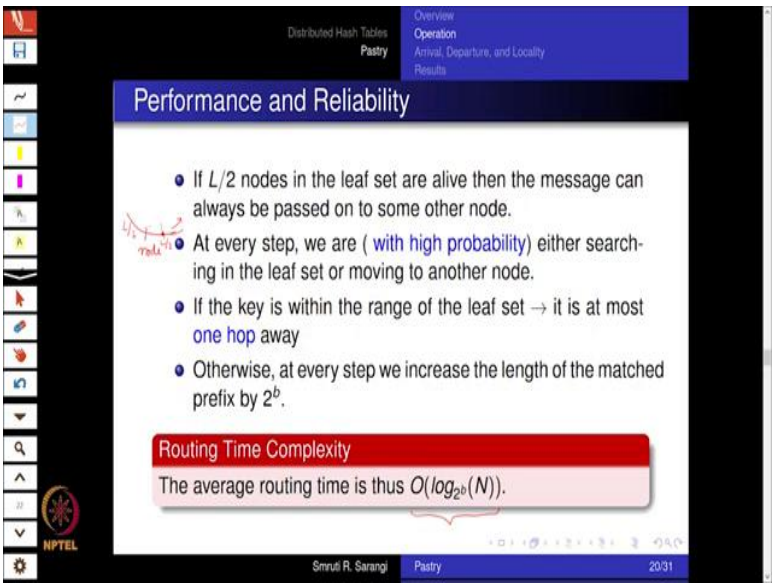
So what you see is that in every step we either increase the match by 1, if we are not able to increase the match then what we do is that at least we are able to reduce the distance and we are getting guaranteed to do that because we have a leaf set, so we are always able to reduce the distance, until the key starts falling within the leaf set of a node. Once the key falls anywhere we know for sure which node it should be, which node owns the key, because our view of the leaf set is perfect and then the message can be passed on to that node.

(Refer Slide Time: 61:42)



So the routing algorithm was simple the node first checks to find if the keys within the leaf set. If so, it forwards the messages to the closest node. Otherwise, Pastry forwards the message to a node with one more matching digit. If both the criteria are not there then at least we do not sacrifice the prefix length, but we send it to another node, which is numerically closer and we are always guaranteed to find such a node. So because of the leaf set we are always guaranteed to find such a node which is closer.

(Refer Slide Time: 62:17)



So the performance and reliability, well, if L/2 nodes in the leaf set are alive, then what you can see is that, look, if this is the current node we have L/2 nodes of the leaf set on either side,

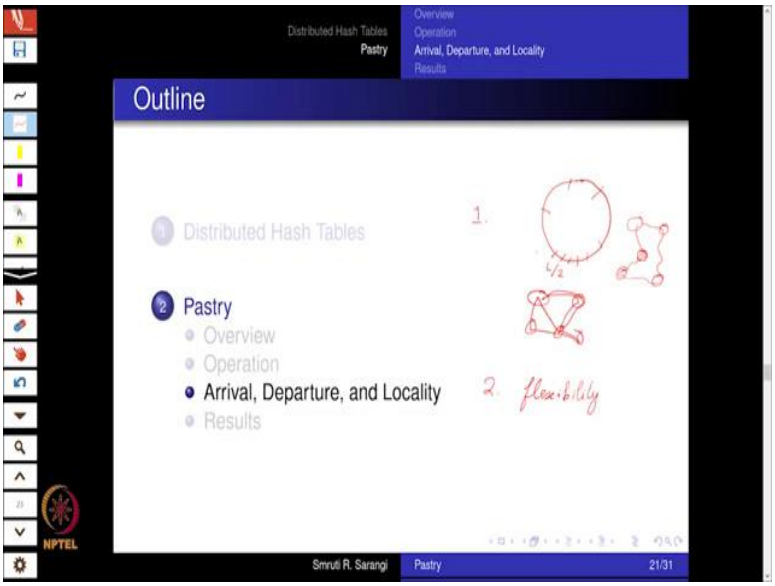
if one among them is alive, then at least the message can be passed on to that and this is guaranteed to be closer if you are traversing in this direction. Again in that node, for that node one among each L/2 nodes on either side is alive, then we can pass on the message to that.

So we always have forward progress. So essentially what this protocol is trying to see is that as long as L/2 nodes on either set, at least one of them is alive we are guaranteed to reach the destination. So, I mean, in that sense there is some amount of built-in reliability and given the fact that the nodes in the leaf set are proximal, proximate in terms of IDs and not in terms of geographical distance.

We can have these nodes physically located in different data centers such that all of them will not suffer from an outage at the same time. So as we said what do we do? We either increase the prefix length or bring the node closer to the key in every step. So the average routing time, so it can be proven with more rigorous methods, but definitely the prefix length match will be log n steps.

And even without that, so even after that when we are traversing the leaf sets, it can be shown experimentally that the average routing time is order log n, O of log n and if you want to be slightly more precise that is O($log_{2^b}(N)$).

(Refer Slide Time: 64:29)



So let us now look at the arrival, departure and the locality aspects of nodes. So it one important point is due here, so one important point that is due here is that when I am creating a large node in a large ring of nodes, given the fact that they are actually arranged by their node IDs and not

by any other metric. It is highly likely that if one node is over here, all the other nodes that are beside it are actually in other locations.

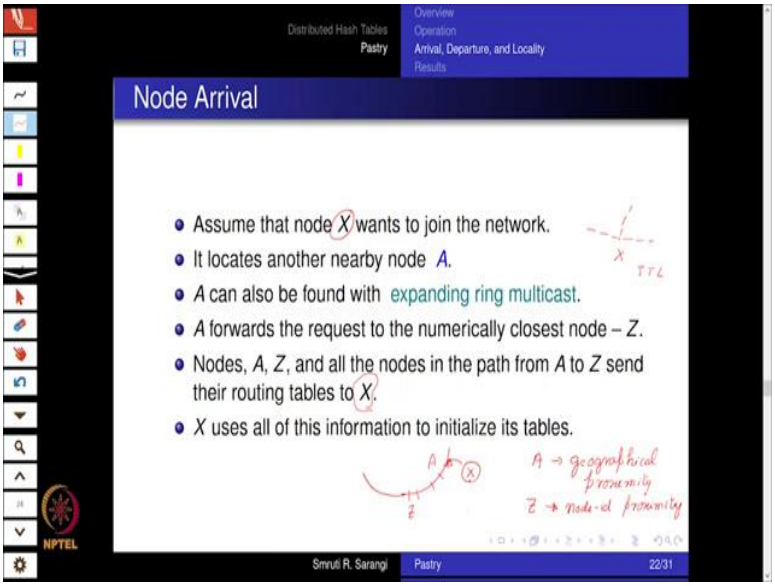
So I would typically create a DHT out of many data centers or a many server farms and the server farm would be located kind of all over the world. You consider Amazon, Facebook and so on, they have lots and lots of server farms that are located all over the world, all of them are kind of connected together in a complicated snake like DHT. And of course, I am kind of drawing a simplified figure but it is way more complicated if I want to do it more realistically.

Well, you might have one connection here that goes here, that goes here, again comes here, again goes there, again here, again there, it does not matter, it is still logically arranged. So the reason for this is that we want L/2 nodes on either side of every node to be alive, one among them to be alive and one among it will be alive, if let us say we spread all of them out geographically.

So a logical ring allows us to do that. Another beautiful property of a DHT is that if you want to add new servers all that we do is we just expand the overlay and we just add the servers, so servers will get added at different points, so this can automatically expand to cater to higher demand. Like e-commerce sites during Christmas, so automatically what will happen is that your DHT can expand and you can add more nodes seamlessly.

And then you can delete nodes same way, the deletion is just an opposition, is just an opposite of addition. So this provides the flexibility that many of these e-commerce sites and movie providers actually require because they can add servers and they can delete servers.

(Refer Slide Time: 67:04)



So the addition protocol is simple. It says that assume that node X wants to join the network, a new node. So it locates a node that is close to it in terms of geographical proximity, in a sense it can be another node in the same data set, in same data center, in the same subnet. Basically another node in the same area, which is close by, it is not on the other side of the world.

If A cannot be found, well, we can do expanding ring multicast, which you would have already seen in the gossip protocol, which essentially means that node X sends a message to its 1 hop neighbors asking them if they are a part of the DHT. If not it sends a message to its 2 hop neighbors asking them if they are a part of its DHT, if not it sends a message with to its 3 hop neighbors.

So this can easily be done with a TTL field of the TCPIP protocol, the IP protocol specifically, where we see that the message will be alive for only two hops or three hops, so with this it is possible to find a node that is closest to you that is a part of the DHT and many DHTs also have centralized directories that maintain a list of nodes that are close to a node that wants to join.

So then, let the closest node be Z, so A forwards the request numerically closest node which is Z. So then what's the idea? Well, the idea is that I have a ring, node X wants to join, it contacts node A which is the closest to it in terms of geographical proximity. Node A then finds node Z, which is the closest to node X in terms of proximity of node IDs.

Then nodes AZ and all the nodes in the path from A to Z kind of cooperate and collaborate and update their routing tables to incorporate information that node X has joined. So what is node A? Well, node A is the closest in terms of geographical proximity or it also can be any node, but geographical proximity is preferred. We will see why and node Z is the one that is closest in terms of node ID proximity.

So locating node Z is node A's job, once node Z has been located, then all the nodes from the path A to Z kind of collaborate and update the routing tables to record the information that X is now in the system. So this is, let us call it node ID proximity. So it is important to remember this fact that X is a node that is joining. It first contacts A, which is the closest. A then contact Z, which is the closest in terms of node IDs and all the intermediate nodes including A and Z, they the nodes from A to Z basically collaborate in this job of adding X to the system.

(Refer Slide Time: 70:49)

So how do we initialize the tables? Well, since A is close to X geographically, X simply copies A's neighborhood set because A’s neighborhood will be X’s neighborhood. Any node that is close to A will be by definition close to X. Say X will just copy A’s neighborhood set. Similarly, Z is the closest to X in terms of node IDs, so X will just use Z's leaf set to form its own leaf set.

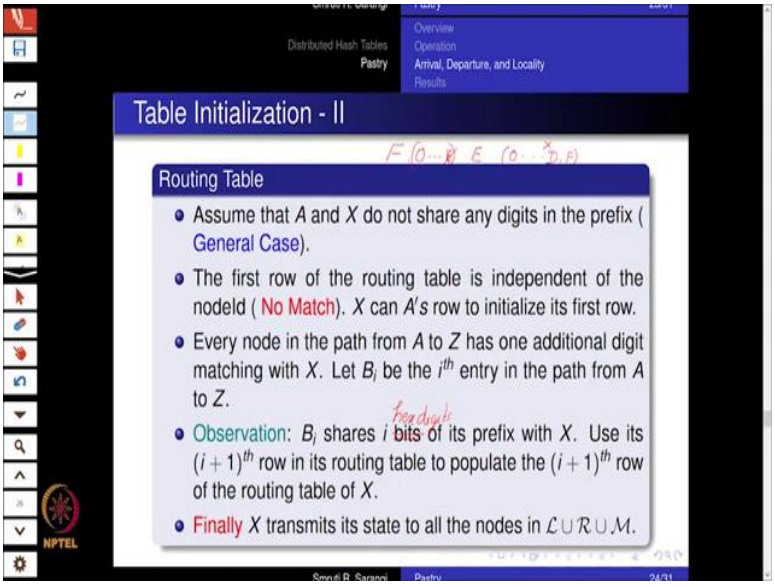
Of course, it will just slightly adjust it, so depending upon whether X is to the left or right of Z, see if this is Z over here and X is over here, then what it will do is that it will form a leaf set like this by taking data from Z's leaf set only and it will also form a leaf set to the right by again taking data from Z's leaf set and doing a few minor additions.

For example, the last node of A's leaf set will not be there in Z's leaf set, so it might have to contact a few more nodes to get more information. So it does not matter how it is exactly done, but the important point to keep in mind is that the neighborhood set will come from A and the leaf set will come from Z. So maybe one important point I just slightly missed out, so it is high time my sort of look at it again.

So how is that located in the first place? Well, that is not hard at all, so X has a node ID, this node ID is given to A. A treats X's node ID as a key. So what A would do is that the node ID that X is giving it, that node ID is treated as a key and that key is looked up at the DHT and then it finds the node that is the closest to that key, which will be Z and that key, since it was X's node ID Z will be the closest to X.

So this as you can see is lo and behold just the key lookup operation. In this case it is not a key but it is actually X's node ID. That is how we get Z and the nice thing is that a neighborhood set will come from A and the leaf set will come from Z, but as I said some minor adjustments need to be made because the leaf sets are not exactly equal. Most of it, there is a huge overlap but they are not equal, that needs to be kept in mind, they are not the same.

(Refer Slide Time: 73:28)



So let us now look at the most important question which is how to initialize the routing table? So if A and X, their first i digits match, well, nothing better. We just take the first i digits, the first i rows and we just copy them from the routing table of A to X. So let us just assume the more general case, the more difficult case where A and X do not share any digits in the prefix.

So that is the more difficult one and if it is all that, the other ones automatically follow. So this means that the first row of the routing table is independent of the node ID. There is no match in terms of digits, but that is okay. The first row of the routing table in any case is supposed to indicate, where certainly no digit matches. So let us say for the node ID, A the first digit is F, and for X the first digit is let us say E.

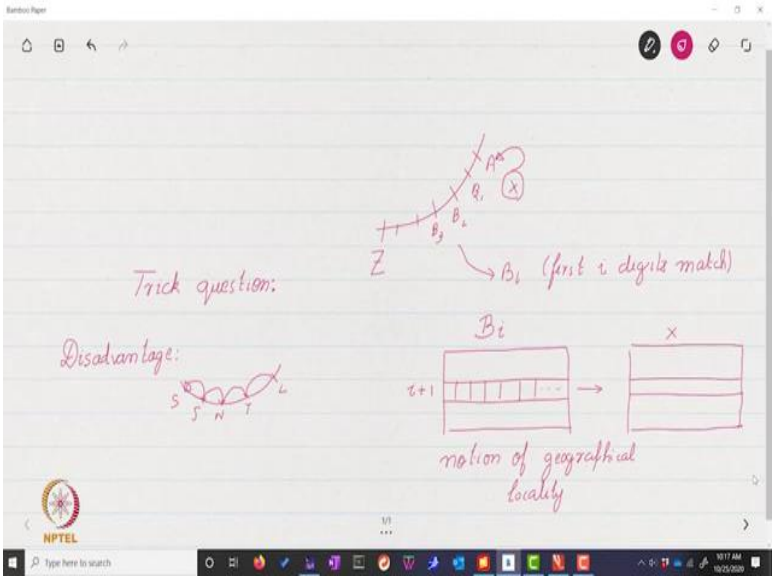
So A will pretty much contain pointers to all nodes whose first digit is from 0 to E, if it does not have a pointer to a node whose first digit is E, then now we have a node whose first digit is E which is X, so that can be added to A's routing table, so that repair process can be done right here. But well we can discard the E part over here and for the first row of the routing table of X, the we can get pointers to all the nodes whose first digits are 0 to D, E is of course not required.

And the first digit and the node whose ID is first digit is F, that is actually node A and that can be added to the routing table of X. So what X can do is it can take the first row of the routing table of A, make a minor modification as was just discussed and populate its own routing table. Then every node in the path from A to Z will have one additional digit matching with X. So let Bi be the i'th entry in the path from A to Z.

So Bi shares i bits of the prefix with X. So what we can do is that we can use the $(i + 1)^{th}$ row of the routing table of Bi to populate the $(i + 1)^{th}$ row the routing table of X. So recall that the $(i + 1)^{th}$ row of the routing table essentially means that the hex digits, see the $(i + 1)^{th}$ row essentially means that the first i hex digits of its prefix match, with the key or whatever we are trying to look up and the $(i + 1)^{th}$ row essentially contains all those digits that do not match. So we have 15 such choices.

So what we can do is that if there is a match for the first i, we just access the $(i + 1)^{th}$ row and copy the routing table from the Bi'th node to X and make small modifications as was discussed in the first, in the case of A and X. So this process continues. So X can essentially use the information from the path from A to Z, so this I will explain slightly graphically as well.

(Refer Slide Time: 77:28)



So if we look at it we had node A which was contacted first, node A took X' node ID and got node Z. Then it sent it through a set of nodes to Z. So this was the message was sent via a set of nodes, and let us call them B1, B2, B3, and so on. So any Bi on this path basically means that the first i digits of the prefix match. This is what it means, so the first i digits. So this is the precise meaning, that first i digits match.

So then what we do is that we look at Bi and we take a look at the $(i + 1)^{th}$ row of its routing table. So this is basically meant to contain pointers to all other nodes, whose $(i + 1)^{th}$ digit is one among these 15 columns. So what we can do is that we can transfer this, transfer this row to the routing table of node X and the transfer is guaranteed to be correct. The reason for this is that the first i digits match, so the $(i + 1)^{th}$ row will contain the pointers for an additional match.

And hence, the information across Bi and X will be the same. But of course, since Bi and X, the digits do not match, one or two of these entries will not be relevant, but those small corrections can be made. So as we proceed from A to Z, we will get all the rows from the routing tables that are relevant and all of them can be added to the routing table of X. So it will now have a fairly reasonably well populated routing table.

So now we come to the trick question? Why do we need a neighborhood set? so the question is that what was the reason, the need for us to choose a node A which is in the kind of the proximity region of X and then why did we go from A to Z? What is the advantage? Well, the advantage is like this that we need to actually start from a disadvantage.

So one disadvantage of DHTs is like this that let us say that I have a key and the node that owns the key is just, it is close by, let us say is in the same data center. So what would happen in the DHTs since we proceed by node IDs and not by actual physical locations? Well, what I am going to do is I am going to jump, jump, jump and ultimately come to the node, but so this process of sending messages, it might take me throughout the world.

It might take me halfway around the world in the sense that I might give a request in London and the next server might be in Tokyo, the next one might be in New York, the next one might be in Stockholm and the final one might be in Sydney, the final node. So the network latency in this case will be high, so I still want to have the advantages of reliability, but I would also like the messages to sort of go in proximate neighborhoods such that the messages also kind of travel fast.

So because of that what we did is we did a small trick in the sense that we start from node A, which is close to X in terms of geographical proximity and furthermore, given the way that you look the way that we have populated these tables, we can kind of assume by induction that when A would have been added, A would have contacted some other node in this which is close by and essentially when that node would have added it would have contacted some other node which is close by.

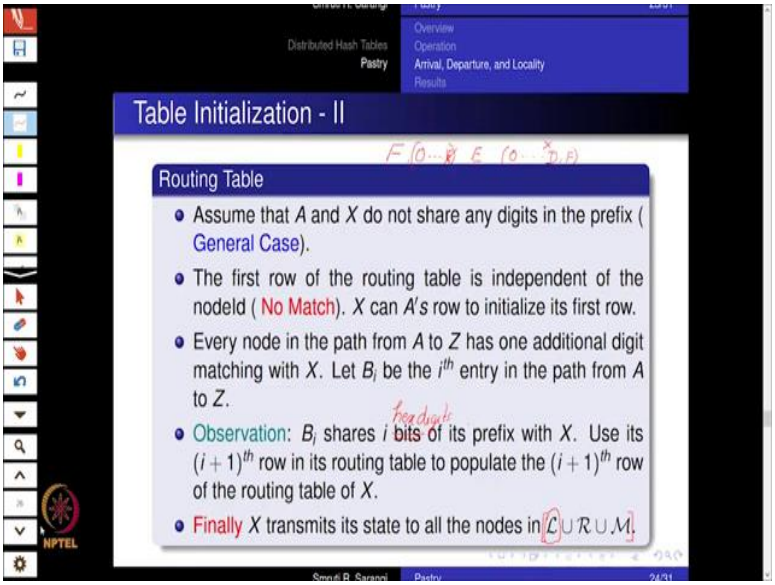
So pretty much all the entries of the routing tables point to nodes that are in the proximate neighborhood. So the entries of the routing tables fundamentally capture a notion of geographical locality. And this is ensured by the way that we actually construct the DHT. So a notion of geographical locality is kind of already captured. Why? Well, because when any node joins, it will always start with contacting a node that is close by.

And that node will then look at its routing table but its routing table would anyway consist of nodes that are close by because typically in the node space we will have many nodes that have matching prefix digits and so on, but out of that we should choose the one that is close by to kind of reduce latency and this as you can see is kind of being ensured by construction. The way that we are constructing this that we always contact the node that is closed by.

It will always route the message among its close contacts, then those nodes will route the message among their closed contacts. So with this itself we are seeing that all the nodes on the path are relatively expected to be geographically closer to X. It will not be totally random. So which means that we are not going to crisscross the world while sending a message from A to

Z. So that is an important point to keep in mind that the neighborhood is handy in this case. It is a performance enhancing technique.

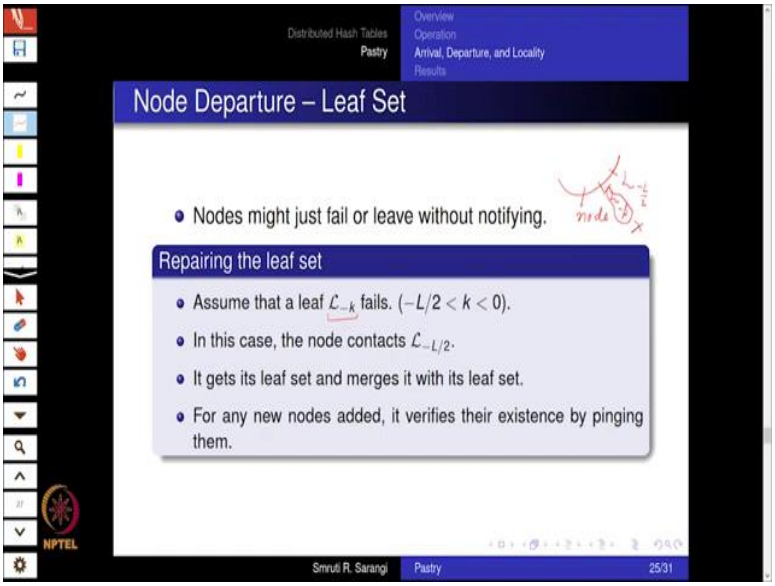
(Refer Slide Time: 84:28)



So let us now come to the last point of this slide is that once X has been added to the DHT it transmits, its states, it makes itself known to everybody in its leaf set, all the nodes in its routing table and all the nodes in its neighborhood set such that they can also update their internal state to take X into account, because as we have mentioned the leaf set particularly needs to be very accurate. So the leaf set has to, the nodes on the leaf set have to update themselves.

The rest of the nodes on the routing table and neighborhood set can gradually, lazily do it, but all of them essentially have to record the fact that X is in the system and X has been fully added.

(Refer Slide Time: 85:23)

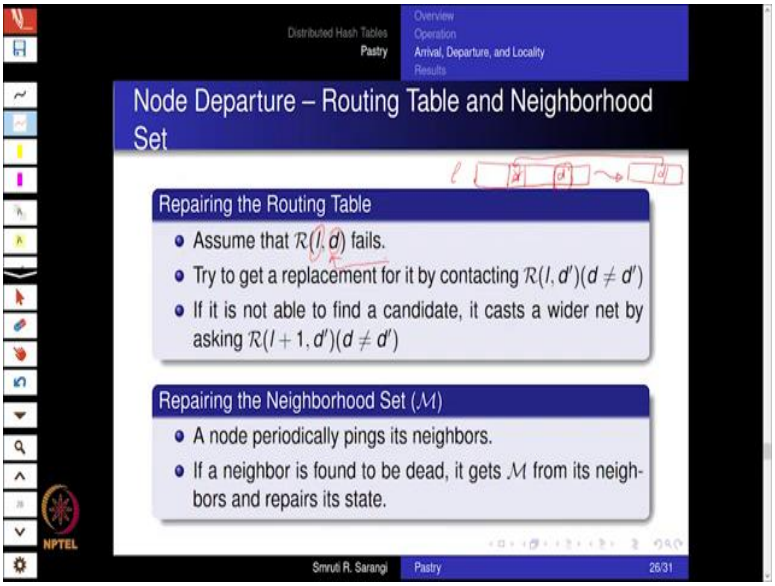


So node departure, well, nodes might leave with or without notifying, say if a node is voluntarily leaving, it might let other nodes. No, otherwise it is possible that it just fails. So, if it fails it will just fail, it will not let anybody know. So assume that if L_{-k} with ID -K fails, which means that it is on one side let us say that it is on the anti-clockwise side.

So in this case once the node is aware of it, so the node can send periodic heartbeat messages to the leaves and once a node is aware that a leaf has actually failed, well, then it needs to reconstruct its leaf set. So it can contact this node, which is at the end of its leaf set. So if this node fails, it can contact the node at the end of its leaf set, get an idea of its leaves and then reconstruct its leaf set accordingly.

So it will essentially merge its leaf set and reconstruct. So what is the key idea? Well, the key idea is that every node keeps a track of its leaf set by sending heartbeat messages, whenever a leaf kind of dies and this fact is discovered other nodes on the leaf set are contacted, they send their leaf set information to the node and the nodes in the vicinity reconstruct their leaf set.

(Refer Slide Time: 87:07)



Repairing the routing table, well, if a node leaves, then let us assume that at row L this entry fails, so let us try to get a replacement for this entry. So let us contact some other entry on the same row. So it is guaranteed that the prefix match, the number of prefix digits that are matching on the same row will be the same. So let us contact another entry.

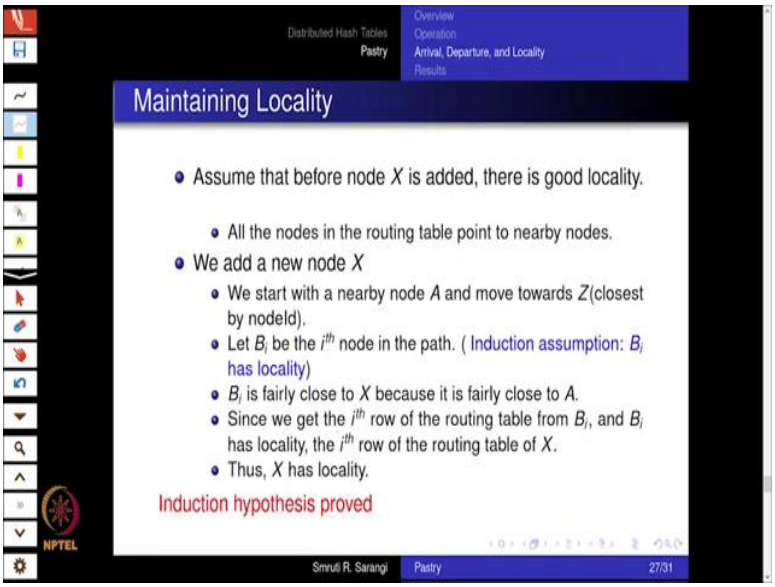
So that other entry, that knows of some other node, which satisfies this criterion, then fine, it will send that to the routing table and the routing table will update itself. But if you are not able to hold on, before saying that I would like to maybe show this graphically, so let this be the l'th row and let us assume that the entry for digit d fails.

Then we send a message for entry digit d' and, so basically this would point to a server, so this would point to a server, a message is sent to that, and it is and we asked the server, hey, do you know of any other node whose first L digits match between the first L digits are the same as your node ID? but the $(L + 1)^{th}$ = d. If it has such an entry in its table, which is not the same server that failed, then it transfers that and that is used to update this entry over here.

If we are not able to find a candidate, well, then we cast a wider net. So what we do is that we start asking other servers with a higher degree of prefix match, not a lower degree, but a higher degree of prefix match and we ask them that look, so essentially the message that is sent is, that I had an entry at R, l, d, that server has failed, so do you know of some other server whose first L digits are the same as your node ID and $(L + 1)^{th}$ = d.

So if those other servers have some record of some other node which has this property, they will send it and the routing table can be repaired. How do we repair the neighborhood set? Well, a node periodically pings its neighbor. If a neighbor is found to be dead, well, then what do we do, what we do is we get m from other neighbors and we try to repair the state. Similar to a leaf set repair.

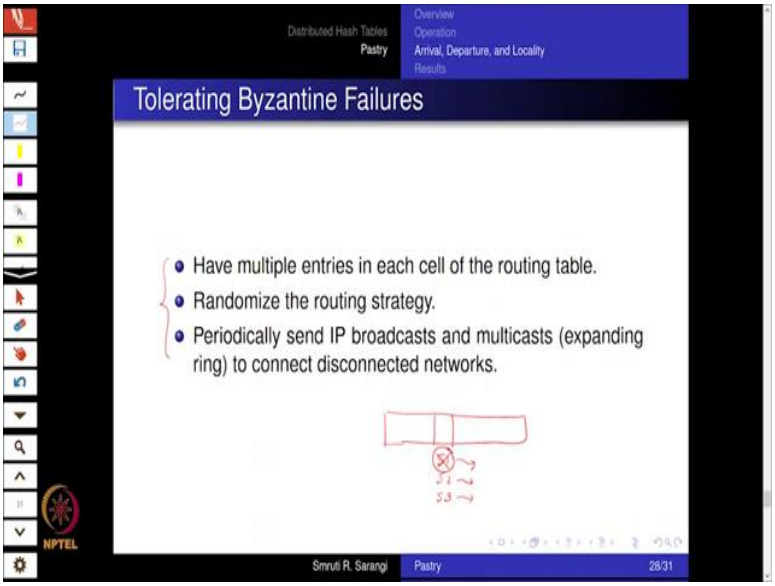
(Refer Slide Time: 90:00)



Maintaining locality, it comes back to the trick question that I had asked a few slides earlier, so assume that before node X is added, there is a good amount of locality. Locality in the routing tables and so on. So this is geographical locality that all the nodes in the routing table point to nearby nodes. So when we add a new node X we start with a nearby node an and move towards Z, we just kind of the closest by node ID.

So let Bi be the i'th node on the path. So the induction assumption is that Bi has locality. So Bi will be fairly close to X because it is fairly close to A and A is close to X. So using a similar logic we can show that all the nodes on the path from A to Z will be reasonably close to X and so this geographical proximity will be maintained, which essentially means that if we were to search for some key on X it will always pass it via nodes that are close to it, so the latency of transmission is lower.

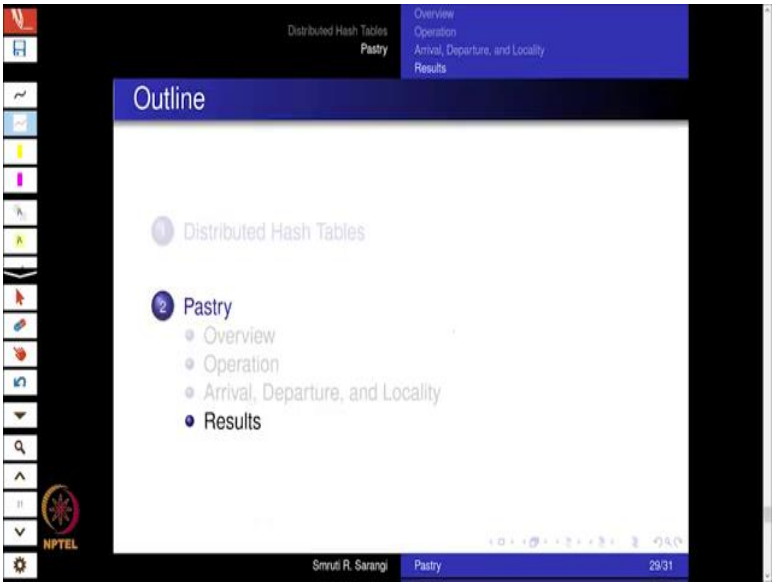
(Refer Slide Time: 91:12)



Can we tolerate more failures, particularly random failures, Byzantine failures where nodes are deliberately malicious? Well, yes. Have multiple entries in each cell of the routing table. So in the routing table instead of having one entry, have multiple entries. We randomize the routing strategy in the sense if we have multiple servers. For some keys we send it to server S1, for some to S2, for some to S3.

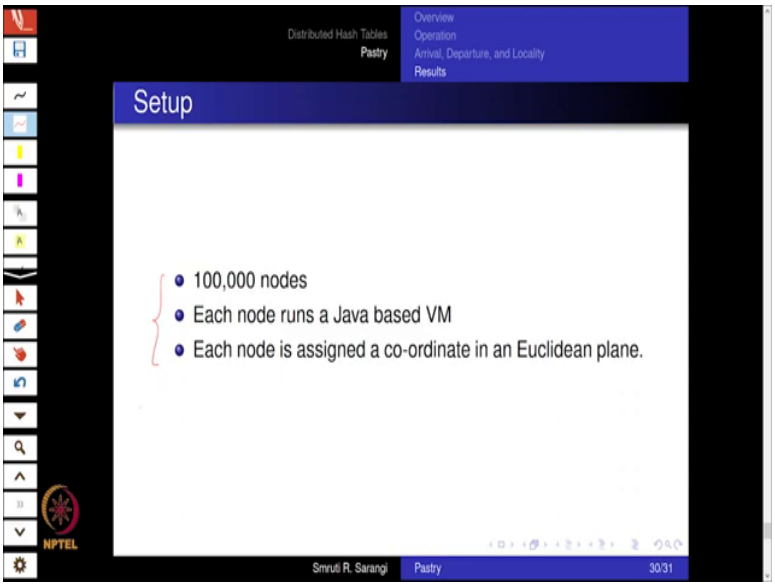
S1 is let us say malicious node, at least the rest of the messages can reach. We periodically try to discover parts of the network by sending IP broadcast, multicast messages, such that let us say if nodes on the leaf set or on the neighborhood set or routing table entries have died, we can again reconnect the network. So we always, so DHTs are in continuous repair mode, like our immune systems, they are continuously checking by pinging other notes and self-repairing themselves.

(Refer Slide Time: 92:26)



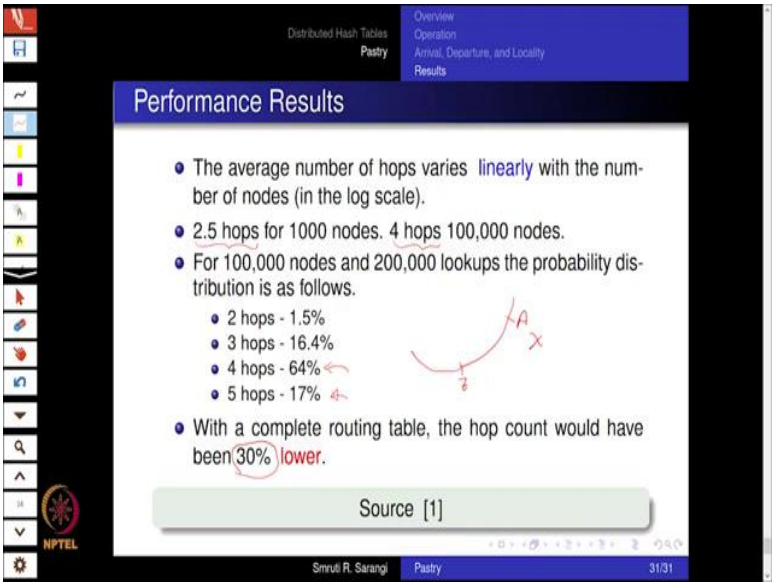
A quick look the results, so the results are, of course, there in the main paper, but we can still briefly look at some key parts of the results.

(Refer Slide Time: 92:40)



So this kind of old paper, so modern DHTs are much bigger, but this was a simulation that had 100,00 nodes. Each node runs a Java based virtual machine, and it is assigned a coordinate on an, on the Euclidean plane X, Y plane.

(Refer Slide Time: 92:58)



So the average number of hops is varied linearly with the number of nodes, so what we see is that the hop count does follow a log trend. So this is clearly visible in the paper, I am just summarizing the key results that for 1000 nodes we have two and a half hops roughly, or 100,00 nodes 4 hops. So it shows that as the number of nodes scale, the hop count increases in a log scale, which is of course, great news for us.

And for 100,000 nodes and 200,000 lookups, the probability distribution of the hops is like this, that one and half percent two hops, most of them are between 3 and 5 hops, 3 hops = 16.4 %, 4 hops = 64% and 5 hop = 17%. So, well, one important point that needs to be kept in mind is that in a DHT, we never have a complete global view of the network that is because we always have a rather local view.

X contacts node A, node A creates a path to node Z, that is how X gets to know about, at least, some part of the network. So the routing tables are by definition incomplete, because for such a large network finding a global view is hard. So, if let us say I somehow have a global view and I am able to create a complete routing table, which has all the information that is needed, not that is needed but that is there in the system.

Then of course, my routing tables will be much bigger and the hop count will lower, but not substantially lower. That is an important point to keep in mind, that the hop count will only be 30% lower, that is what the paper says and this, if you look at it is actually a fantastic result, so it shows that even with a limited amount of information, we are able to do quite well. So we may not have a global view of the network.

So global view of the network is feasible only for small networks, but for large networks, we kind of look at a small region and we do our job. So this does imply a sense of global optimality, not optimality but we can say that it is not that bad. A 30% reduction in average hop count is okay, is something that is tolerable, not a reduction, a 30% increase is tolerable given the fact that we have a practical protocol.

(Refer Slide Time: 95:49)

So the original Pastry paper was published by Antony Rowstron and Peter Druschel, in Middleware 2001. So I would request all viewers to read the paper. The paper has all the details of the experiments, the protocols, everything. And of course, if somebody really wants to learn Pastry, the best way is to implement the Pastry algorithm using a popular language like Java or Python, and actually see it running.

That is the best way to learn this. Thank you very much for bearing with me in this long lecture. So next we will discuss a quad DHT, which is conceptually similar to Pastry in the sense it does have the same concepts, but it is architected very differently.